REPORT DOCUMENTATION PAGE

AD-A239 419

Form Approved
OMB No. 0704-0188

2

T DATE	3. REPORT TYPE AND DATES COVERED
August 1991	final report 30Aug88-28Sep90

4. TITLE AND SUBTITLE

Mathematical Theory of Computation

5. FUNDING NUMBERS

C: N00039-84-C-0211
T: 20

6. AUTHOR(S)

John McCarthy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

sponsoring agency:
SPAWAR 3241C2
Space & Naval Warfare Systems Command
Washington, D.C. 20363-5100

monitoring agency:
ONR Resident Representative
Mr. Paul Biddle
Stanford Univ., 202 McCullough
Stanford, CA 94305

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release: distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

See attached report.

91-07580

14. SUBJECT TERMS	15. NUMBER OF PAGES
	6
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UL	UL	UL	UL

Sponsored by

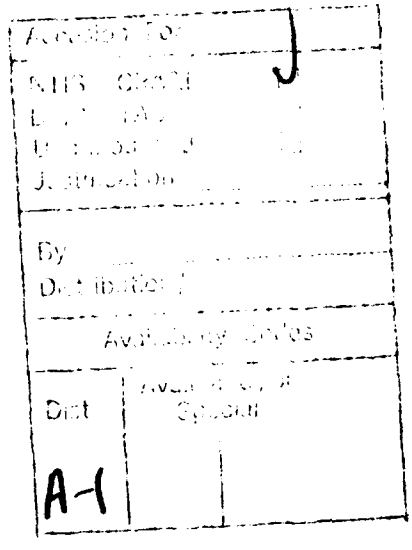
Defense Advanced Research Projects Agency (DoD)
3701 North Fairfax Drive
Arlington, VA 22203-1714

"Mathematical Theory of Computation"

ARPA Order No. 6116

Issued by Space and Naval Warfare Systems Command

Under Contract No. N00039-84-C-0211, Task 20



Final Report for Task 20 of Contract N00039-84-C-0211

Programming and Proving with higher order abstractions and reflection

Project Summary

This project was concerned with the development of correct and reusable software through the use of higher order abstractions (function, control, assignment, process) and reflection. A semantic framework for these notions will be the basis of an experimental system for manipulating and reasoning about programs.

The goals of this project were the development of logical formalisms for reasoning about programs that use abstractions and reflection, and the application of these theorctical results to selected software problems. Example applications include (1) clarification of existing programming paradigms, (2) analysis of existing and proposed languages used in the DARPA community for specifying, writing, and transforming programs, and (3) development and implementation of tools for computer aided reasoning about and operating on programs.

The accomplishments of this project fit into four categories:

- logics for reasoning about function and control abstractions;
- logics for reasoning about data mutation;
- logics for reasoning about function and control abstractions in the presence of mutable data;
- applying methodology for reasoning about programs to the mechanical verification of hardware.

1. Logics for Function and Control Abstractions

The results of this research are describe in detail in the publications [15, 16].

The first of these, [15], is based on lectures given at the Western Institute of Computer Science summer program, 31 July - 1 August 1986. Here we focus on programming and proving with function and control abstractions and present a variety of example programs, properties, and techniques for proving these properties. Examples include such powerful programming tools such as functions as values, streams, aspects of object oriented programming, escape mechanisms, and coroutines. We begin with an intensional semantic theory of function and control abstractions as computation primitives. A first order theory of program equivalence based on this semantics is developed and used to formalize and prove extensional properties of programs. In addition a method is developed for transforming intensional properties of programs into extensional properties of related programs called derived programs. Application of this method to formalize and prove intensional properties of programs.

In the second, [16], a theory IOCC (Impredicative theory of Operations, Control, and Classes) is presented. This paper represents the initial stage of a project to develop a wide

spectrum formalism which will support not only reasoning about program equivalence, but also specification of programs and data types, reasoning about properties of computations, operations on programs, and operations on program specifications. IOCC is a variant of Feferman's theories of operations and classes [5, 6, 7] and is most closely related to IOC_λ (without the ontology axiom) [7]. The main difference being our choice of basic constants and axioms for equivalence. Program primitives include functional application and abstraction, conditional, numbers, pairing, escape, and continuation capture and resumption. The theory of equivalence is based on the semantic model presented in [15]. In [16] we have simplified the notation, refined the axioms, and eliminated the dependence on certain extensionality axioms which are valid in the model of [15], but fail in the presence of memory effects. In addition, the paper demonstrates that rigorous, but informal reasoning such as appears in [15] can be carried out naturally within a Feferman style theory. The theory of classes is just that of [6, 7]. What is new here, beyond a few class constants specific to our programming language, is the application to reasoning about control abstractions, and the use of maximum fixed-point constructions for reasoning about streams and coroutines. Examples are given for introduction of escape mechanisms into programs by transformation, and for specifying streams and coroutines.

The call-by-value lambda calculus lies at heart of most programming languages. For these languages tools such as compilers, partial evaluators, and other transformation systems often make use of rewriting systems that incorporate some form of beta reduction. For purposes of automatic rewriting it is important to develop extensions of beta-value reduction and to develop methods for guaranteeing termination. In [8] we describe a simplifier for such languages based on the applicative axioms of the theory IOCC. The main innovations are (1) the use of rearrangement rules in combination with beta-value conversion to increase the power of the rewriting system and (2) the definition of a non-standard interpretation of expressions, the generates relation, as a basis for developing terminating strategies.

2. Logics for Data Mutation

The main accomplishment in this area was the development of a syntactic approach to semantics. This new approach was the key insight needed to establish the completeness of an inference system for reasoning about data mutation. The results of this research are describe in detail in the publications [10, 11].

In [11] this paper we study the constrained equivalence of programs with effects. In particular, we present a formal system for deriving such equivalences. Constrained equivalence is defined via a model theoretic characterization of operational, or observational, equivalence called strong isomorphism. Two expressions are strongly isomorphic if in all memory states they return the same value, and have the same effect on memory (modulo the production of garbage). Strong isomorphism implies operational equivalence. The converse is true for first-order languages; it is false for full higher-order languages. Since strong isomorphism is defined by quantifying over memory states, rather than program contexts, it is a simple matter to restrict this equivalence to those memory states which satisfy a set of constraints. It is for this reason that strong isomorphism is a useful relation, even in the higher-order case.

The formal system we present defines a single-conclusion consequence relation between finite sets of constraints and assertions. The assertions we consider are of the following two

forms: (i) an expression fails to return a value, (ii) two expressions are strongly isomorphic. In this paper we focus on the first-order fragment of a Scheme- or Lisp-like language, with data operations 'atom' 'eq' 'car', 'cdr', 'cons', 'setcar', 'setcdr', the control primitives 'let' and 'if', and recursive definition of function symbols. A constraint is an atomic or negated atomic formula in the first-order language consisting of equality, the unary function symbols 'car' and 'cdr', the unary relation 'cell', and constants from the set of atoms. Constraints have the natural first-order interpretation.

Although the formal system is computationally adequate, even for expressions containing recursively defined functions, it is inadequate for proving properties of recursively defined functions. We show how to enrich the formal system by addition of induction principles. To illustrate the use of the formal system, we give three non-trivial examples of constrained equivalence assertions of well known list-processing programs. We also establish several metatheoretic properties of constrained equivalence and the formal system, including soundness, completeness, and a comparison of the equivalence relations on various fragments.

3. Higher-order Abstractions in the Presence of Mutable Data

Reasoning about programs in languages with both higher-order abstractions and mutable data presents problems not present when treating these features separately. The key result in solving these problems was a simple characterization of operational program equivalence in this language. This results of this research are reported in [9, 13, 14, 12]

The key methods developed in [9] consisted in establishing program equivalence by computation induction based on our simple characterization of operational equivalence. Progress has been made in finding a small collection of rules that comprise the main uses of computation induction and to develop further syntactic methods for conditional reasoning. In [13] we describe a *simulation induction* principle that can be used to establish equivalence of higher-order objects with mutable local store. In [12] progress towards a theory of program development by systematic refinement is described. Here a formal system for propagating constraints into program contexts is presented. In this system, it is possible to place expressions equivalent under some non-empty set of constraints into a program context and preserve equivalence provided that the constraints propagate into that context. Constrained equivalence and constraint propagation provide a basis for systematic development of program transformation rules. Three key rules are: subgoal induction, recursion induction, and the peephole rule.

In [17] we take a look at partial evaluation from the point of view of symbolic computation systems, point out some challenging new applications for partial evaluation in such systems, and outline some criteria for a theory of partial evaluation. The key features of symbolic computation systems are summarized along with work on semantics of such systems which will hopefully aid in meeting the challenges. The new applications are illustrated by an example using on the concept of component configuration. This is a new idea for software development, based on the use of higher-order and reflective computation mechanisms, that generalizes such ideas as modules, classes, and programming in the large. In [14] we report progress in development of methods for reasoning about the equivalence of objects with memory and the use of these methods to describe sound operations on such objects, in terms of formal program transformations. This work combines the methods

developed in [13, 11, 12] and meets some of the foundational challenges implicit in [17]. Three different aspects of objects are formalized: their specification, their behavior, and their canonical representative. Formal connections among these aspects provide methods for optimization and reasoning about systems of objects. To illustrate these ideas a formal derivation of an optimized specialized window editor from generic specifications of its components is given. Components, or objects, are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In our framework objects are represented as functions (closures) with mutable data bound to local variables. The techniques for reasoning about objects include: rules for establishing equivalence under a set of constraints; symbolic evaluation with respect to a set of constraints; propagation of constraints into program contexts; the method of simulation induction, used to establish the equivalence of objects. The key new result presented in this paper is the (**abstractable**) theorem. This result enables one to make use of symbolic evaluation to establish the equivalence of objects. In the current state of development the framework treats only sequential computation. However, the techniques such as simulation induction and constraint propagation, have been designed with the goal in mind of treating objects which exist in and communicate with other objects in an open distributed system.

For the transformational approach we focused on methods for proving equivalence of definitions, as well as establishing invariants (partial correctness statements) needed to prove equivalence. For this purpose subgoal induction and recursion induction are appropriate. Treatment of total correctness requires the formulation of principles for induction on well-founded orderings. Examples of structural induction principles extended to the case of computations with effects are given in [11] and additional principles have been formulated to treat cases where the measure that is being decreased is not a simple structural property, but decreases due the effect that is produced by the computation.

Recently we have extended the equational theory to fully quantified modal language. This incorporates both constrained equivalence and constraint propagation into a single uniform language. Work is in progress to further extend the language to a Feferman style theory of operations and classes [5, 6, 7]. This extends the work of [16] and will provide a rich language for defining constraints and basis for studying types and equivalence in a uniform framework.

4. Mechanical verification

Hardware verification is an important application area for mechanical theorem proving. In this research the basic approach was to treat circuit descriptions as programs and apply methods of the above programming language theory to develop a formal semantics (interpreting circuits as computing functions on finite strings). The semantic theory was mechanized in the Boyer-Moore logic and a variety of circuit properties were mechanically checked. Viewed as programs, circuits often quite simple and mechanical verification of functional properties seems quite feasible. The methods developed in this work for carrying out the mechanization will provide further basic tools for mechanical verification of program components. The results of this research are reported in [2, 4, 3, 1].

In [2, 1] a new functional semantics is proposed for synchronous circuits, as a basis for reasoning formally about that class of hardware systems. Technically, we define an extensional semantics with monotonic length-preserving functions on finite strings, and an

intensional semantics based on functionals on those functions. As support for the semantics, we prove the equivalence of the extensional semantics with a simple operational semantics, as well as a characterization of th circuits which obey the "every loop is clocked" design rule. Also we develop the foundations in complete detail, both to increase confidence in the theory, and as a prerequisite to its mechanization. The theory has been implemented in the Boyer-Moore theorem prover [1]. A sequence of synchronous circuits of increasing "sequential complexity" proposed by Paillet were mechanically verified using this implementation of the theory [4, 1]. Several notions of correctness for pipelined designs were formalized and correctness of pipelined synchronous circuits, including in the the Saxe-Leiserson retimed correlator, a pipelined ripple adder, and an abstract pipelined CPU were also verified [3, 1].

5. References

[1] Alexandre Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits.* PhD thesis, Stanford University, 1989.

[2] Alexandre Bronstein and Carolyn Talcott. String-functional semantics for formal verification of synchronous circuits. Technical Report STAN-CS-88-1210, Department of Computer Science, Stanford University, 1988.

[3] Alexandre Bronstein and Carolyn Talcott. Formal verification of pipelines based on string-functional semantics. In *IFIP 1989 workshop on applied formal methods for correct VLSI design*, 1989.

[4] Alexandre Bronstein and Carolyn Talcott. Formal verification of synchronous circuits based on string-functional semantics: The 7 paillet circuits in boyer-moore. In *C-Cube Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989*, 1989.

[5] S. Feferman. Constructive theories of functions and classes. In *Logic Colloquium '78*. North-Holland, 1979.

[6] S. Feferman. A theory of variable types. *Revista Colombiana de Matématicas*, 19, 1985.

[7] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In *Logic and Computation*, volume 104 of *Contemporary Mathematics*. A.M.S., Providence R. I., 1990.

[8] L. Galbiati and C. Talcott. A simplifier for untyped lambda expressions. In *CTRS90*, 1990. to appear as LNCS volume, full version Stanford University Computer Science Department Report STAN-CS-90-1337.

[9] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[10] I. A. Mason and C. L. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Department of Computer Science, Stanford University, 1989.

[11] I. A. Mason and C. L. Talcott. Inferring the equivalence of (first-order) functional programs that mutate data. *Theoretical Computer Science*, to appear, 199?

[12] I. A. Mason and C. L. Talcott. Program transformation via constraint propagation, 1990. to appear.

[13] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, to appear, 1991.

[14] I. A. Mason and C. L. Talcott. Program transformation for configuring components. In *Symposium on Partial Evaluation and Mixed Computation, PEPM'91*, pages 297–308. ACM, 1991.

[15] C. L. Talcott. Programming and proving function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department, 1989.

[16] C. L. Talcott. A theory for program and data specification. In *Design and Implementation of Symbolic Computation Systems, DISCO'90*, volume 429 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. full version to appear in Theoretical Computer Science.

[17] C. L. Talcott and R. W. Weyhrauch. Partial evaluation, higher-order abstractions, and reflection principles as system building tools. In D. Bjorner, A. P. Erschov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North–Holland, 1988.